CHAPTER 3

ARCHITECTURE FOR MANAGING RESOURCE USAGE POLICIES

3.1 OUTLINE

In this chapter, the proposed architecture for managing RUPs and SLAs are explained in detail. In addition, the components of PMS that includes policy editor to create, modify and delete the RUPs are explained precisely. Besides, the GSMA architecture to create, monitor and enforce the accepted service levels is elucidated elaborately. The PMS and GSMA are explained in this chapter, whereas the Deviation based Resource Scheduling is described in next chapter.

3.2 PROPOSED ARCHITECTURE

Grid is a dynamic framework that enables sharing and selection of resources over geographically dispersed locations; a high performance computing environment that comprises scalable fault tolerant, loosely coupled heterogeneous resources across multiple organizations called as VO. A VO is a collaborative computing environment, where a number of resource providers seek to share their resources in order to meet a common objective. Resource sharing in such a collaborative grid environments is a serious issue. Because the resources that belongs to different administrative domains may have different allocation policies in different VOs. The heterogeneity in these policies makes the problem more complex than the fair share allocation in
clusters. Also policies in this context are broad, complex and unclear. So it is mandatory to express and enforce these policies for proper functioning of these collaborative environments. Co-allocation demands the grid meta-scheduler to co-ordinate multiple resource providers located in different administrative domains. The grid meta-scheduler acts as an orchestrator and negotiates with potential resource providers in order to create the resource usage policies dynamically. Currently, these policies have no direct connection with the grid meta-scheduler that controls the entire grid environment. Due to lack of these connections, there is no provision to restrict/control the grid resource sharing that prevents the resource providers in offering their resources to the grid environment. The proposed architecture consists of following components:

- **PMS** to create, modify and to delete resource usage policies that reflects the desired usage scenarios.
- **DRS** that aids in SLA guided resource selection.
- **GSMA** that is capable to negotiate, create, monitor and to enforce the SLAs thereby ensuring that the accepted or desired usage scenarios of resource providers obeyed.

### 3.2.1 Policy Management System (PMS)

Managing grid resources is a complex task since the grid spans across multiple organizations. Every organization has their own resource usage policies expressed using their own mechanism. Whenever the resources from these organizations are contributed to grid environment, it is not possible to set single usage policy on a whole. Because the nature, locality and the business hours of an organization may determine the available time of resources, amount of resources that can be contributed to grid and the desired
usage scenarios. Currently there is no mechanism available to give assurance to the resource providers that their local resources are never overrun. In order to realize the controlled grid resource sharing environment, there must be a common mechanism to express these usage policies for all resource providers participating in grid and these policies should be interconnected with the grid meta-scheduler in order to enforce the desired usage scenario. In general, there are two kinds of policies in grid environment: Access policies and usage policies. The access policies are used for authentication and authorization and the usage policies specify the accepted usage scenario. The policy management system proposed in this thesis is capable of:

- Creating the resource usage policies through the policy editor.
- Adding / deleting the new/existing policies to/from the policy repository.
- Policy matchmaking mechanism that can identify the appropriate policy from a set of usage policies and compares the usage policy against current job requirement.
- Integrating this policy management system with our CARE resource broker.

First, the resource usage policy expression has to view the policies at different levels of grid environment that includes local policies, grid site policies, and operational policies. The local policies mainly concentrate on the resources within a domain. They mainly focus on amount of contribution (such as CPU load, storage) towards the grid environment on every resource that participates in grids. For example the usage policy that specifies “…provide 20% of disk space towards grid on machine A ..” falls under this category. When many resources belonging to different domains combine
together to form a grid, there must be some rules and regulation about the sharing of resources, quality of services to be maintained during the interaction between them. These are specified under grid site policies. These policies are otherwise called as service level agreements. For example, “…the Madras Institute of Technology (MIT) has to provide 60 CPUs to Garuda whereas the Garuda has to provide 100Mbps (Mega bits per second) connectivity to MIT…” coming under grid site policies. These policies mainly concentrate on single site rather than entire operation of grid environment. The policies that concentrate on the entire operation of grid environment are called as operational policies. For example, “…The overall load for VO$_1$ should be less than 70%…” belongs to this category.

After viewing the policies at various levels, the second step is to express the policies using some existing/new policy language. No standard has been followed to express policy information in grid environment to represent the resource usage policies. This research work uses WS-Policy specifications to express policy information thereby making it possible to address interoperability issues across various grids and grid resources. WS-Policy provides a flexible and extensible grammar for expressing the capabilities, requirements, and general characteristics of entities in a XML Web services-based system. A new XML schema is proposed to express the usage policies in grid environment. The usage policy expressed (only local policies) using this schema is embedded into WS-Policy and put into policy repository. Every resource provider can specify one or more usage policies for every resource that contribute to grid environment. The proposed schema for resource usage policy is shown in Figure 3.1.
Figure 3.1 Proposed schema for resource usage policy

Every resource provider has unique identifier called Resource Provider Identifier (RPID) and is specified within Resource Provider tag. Every policy has unique identifier, policy identifier (P-ID) and is specified within Policy tag. The name of the resources bounded by the current ID is specified in the ResourceName tag under the Resource tag. The duration of availability of each resource is specified in the Availability tag. Also if the resource provider willing to allow the users from particular domain, they can
use the allow tag that is available under Security tag. The FileSystem tag is used to specify the mount point, where the user jobs use the space available in this filesystem for executables, input and output files. The number of CPUs, amount of CPU load, bandwidth and the RAM contributed towards the grid environment are specified in the CPUCount, CPULoad, NetworkBandwidth and PhysicalMemory tags respectively.

In order to make the policy creation easy, the above mentioned tags are transformed into a portal, where a resource provider can simply enter the desired usage scenario and submit it to policy manager (refer to Figure 3.2). The policy manager then converts the entries in the portal into a XML file (RSLA.xml) and stores it in the policy repository. Also, there is a portal support to add/modify/delete the new/existing policies from the repository. Whenever the resource provider willing to add a new policy, he/she has to simply login as resource provider and click the addPolicy button. Then the portal will be redirected to createPolicy page wherein the resource provider enters his desired usage scenario and submit it to policy engine to update it in the policy repository. Then the policy engine adds one more Policy element (tag) in the corresponding ResourceProvider tag identified with the help of logged in RPID and updates the RSLA.xml file. If he wishes to modify the existing policy, he has to click the modifyPolicy button, then it will show all the policies created by him one by one. He may use the next navigation key in order to navigate through all the policies and make appropriate changes and add it to the policy repository. In this case, the policy engine updates the corresponding modified policy tag identified with the help of PID in the related ResourceProvider tag tracked by the RPID. The deletion of existing policies can be done through deletePolicy button. Whenever the resource provider clicks the delete button, all the policies related to this particular resource provider are retrieved from the repository and shown to resource provider. Then the user may navigate to the appropriate policy, delete it and update in the repository by removing the Policy tag.
In the third step, find out the appropriate usage policy first and then compare that usage policy against the current job requirements in order to test the suitability of that particular resource. For example, a particular resource provider may define many usage policies for a single resource based on his business hours. Alternatively, the resource provider determines the resource available time and the amount of resources contributed to the grid environment based on their local usage. Whenever the user submits his job to the meta-scheduler, he may want to run the job immediately (i.e. On-demand) or some time latter (i.e. Advance Reservation (AR)). Then the meta-scheduler has to identify the resources that are available immediately or the resources that are available at their Expected Time of Run (ETR) respectively. In both cases, first the meta-scheduler has to identify the appropriate usage policy for all the available resources. In order to select the correct usage policy, it will use the availability of resources specified in the usage policy as its primary key. For example, an user submits the job to the meta-scheduler and expects the job to be run at latter time. Currently there may be 100 resources available for job submission. But there is no assurance that all the 100 resources are
available at the expected time of run. In order to discover the resources that are available at the expected time of run, the meta-scheduler simply refers the available time of the resources that is specified in the usage policy. After referring the usage policy, it may come to know that only 50 resources are available at the desired time of run. So the first level of filtering is done by simply referring the usage policy. Then the next level of filtering is done by match-making these resultant resources’ (i.e. 50 resources) capability against the current job requirement. The next step is to create SLAs by negotiating with these resources. Deviation based resource scheduling algorithm is used to reduce the number of negotiations. The pseudo code for selecting the appropriate policy is shown below. Where \( A = \{ \text{host}_1, \text{host}_2, \ldots, \text{host}_n \} \), \( RUP(A[i]) = \{ \text{RUP}_1, \text{RUP}_2, \ldots, \text{RUP}_n \} \), \( i = 0, 1, \ldots, n \), \( \text{ETR} = \text{Expected Time of Run} \).

### 3.2.2 Grid SLA Management Architecture (GSMA)

Service level agreements (SLAs) are powerful mechanisms for expressing all commitments, expectations and restrictions in a business transaction. In a grid environment, the brokers should have the negotiation support to establish SLAs, which is still lacking in most conventional brokers/metaschedulers. These newly created SLAs clearly express the required QoS to be maintained till the end of application execution, restrictions on resource usage and the penalties in case of failure to provide the accepted QoS or breaching the restrictions on resource usage. After the successful creation of SLAs, the resource provider allows the user to gain access to the resources. These resources should be monitored in order to ensure that the committed QoSs are obeyed. In case of any violation against these committed values, there should be a mechanism to penalize the user or resource provider. Currently, these SLAs have no direct connection with the grid meta-schedulers that control the entire grid environment.
The proposed GSMA supports the entire lifecycle of SLAs and also integrated with grid meta-Scheduler to realize a controlled grid resource sharing. The life cycle of an SLA starts with SLA creation phase. Upon successful negotiation, the SLA is formed in SLA creation phase. The SLA creation phase utilizes the proposed DRS to compute the deviation value, order and select the resources for negotiation. The generated SLA specifies the guarantee terms (QoS) and penalty. The SLA monitoring phase monitors these guarantee parameters specified in the SLA. The SLA violations are notified to the SLA enforcement phase (using delegation event model) that enforces the SLA against the violation and generates necessary actions and entire lifecycle is depicted in Figure 3.3. The Violation notifications and events are fired when: (1) a submitted job over-consumes the guarantee parameters (such as CPU-percentage or RAM) than the value actually committed by the provider-(requestor side fault). (2) When the available value of guarantee parameters (such as CPU-percentage or RAM) for the submitted job less than the value actually committed by the provider- (provider side fault). The above mentioned tasks are grouped together and placed under various components of GSMA and is shown in Figure 3.4.

**Figure 3.3 Lifecycle of SLA**
SLA Creation Engine - The creation phase of SLA lifecycle starts with negotiation that comprises of SLA negotiation and Commit components. The negotiation component is responsible for gathering the commitments of each resource provider, and generates a proposal. The commit is responsible for obtaining the final decision against the proposal thereby converting a proposal into agreement. If the proposal is accepted by all the resource providers participating in negotiation, then it will become an agreement and is stored in a SLA database. If not, re-negotiation process is initiated with the replacement of the resource providers who reject the proposal with new one whose information is getting from Broker. The commit is responsible for agreement creation and storing it in a SLA database.

Negotiation is a methodology by which an agreement is established on a range of objectives based on an interaction between the participants using some strategies. There are two entities involved in a negotiation process: Negotiation protocol and Negotiation strategy. The negotiation protocol specifies set of all possible messages that are involved in a negotiation process irrespective of the negotiation strategy that is used. The
negotiation strategy specifies the policy or methodology used for negotiation. Hence, the flow of negotiation protocol messages differs in different negotiation strategies. Here, the interactions are done through a common negotiation protocol that supports multiple negotiation strategies such as unilateral bargain, bilateral bargain, auction and strict (i.e. no more bargaining is supported). Hence, the proposed approach has a negotiation manager that supports multiple negotiation strategies using a single negotiation protocol. The proposed negotiation framework is shown in Figure 3.5.

![Proposed negotiation framework](image)

**Figure 3.5 Proposed negotiation framework**

The proposed negotiation framework uses double auction (refer to Figure 3.6) based negotiation model, where, ‘n’ resource providers bid against ‘m’ job requests. The requirements of an application are expressed as a tender/proposal and it is submitted to the broker whereas, the resource providers expressed their bid as resource usage policies against each job. Each resource provider can have more than one resource usage policy based on the availability of resources, peak load time, least load time etc. The matchmaking module of the broker selects the resource provider for negotiation only when the resource provider has a favourable resource usage policy against the tender in all aspects at the expected time of job execution. For instance, if a tender needs a job to be executed after five hours from the time of submission, then the matchmaker identifies the appropriate resource
usage policy to be applied for that time (i.e. after 5 hours) and extract the bid from that policy and compare it against the tender. If the bid satisfies the tender then the resource provider will be selected for negotiation. If not, then the resource provider will be selected for negotiation only when their negotiation strategy is other than strict. The latter tries to reduce the length of negotiation process by considering the success probability of bargaining, whereas the former is a certain event.

The modified Mutual Agreement Protocol (MAP) is used for negotiating with the resource provider. It consists of following six messages: Request, Accept, Reject, Re-Request Alternate and commit.

**Request:** This message is initiated by the client, while sending the requirements to the provider.

**Accept:** This message may be initiated either by the client or provider, while admitting an offer.

**Reject:** This message may be initiated either by the client or provider, while refusing an offer and ends the negotiation.
Alternate: This message is initiated by the provider, while providing a counter offer against the original request.

Re-Request: This message is initiated by the client, while sending the altered requirements to the provider.

Commit: This message may be initiated either by the client or provider, while acknowledging an offer and ends the negotiation.

The flow of messages during negotiation of a client/broker with single resource provider is explained in Figure 3.7. The numbers are used to provide better explanation. To start the negotiation process, the Client sends a Request message to the resource provider. It consists of requirements of an application expressed as a tender. Upon receiving the request, the provider may accept to provide the requirements thereby acknowledging the tender/proposal and convert it into agreement. Otherwise, he may reject the tender/proposal and ends up the negotiation process. Alternatively, he may provide counter offers rather than requested one by initiating alternate message. For this, client may create a new request and send it to the provider by initiating the re-request message or initiating the reject message for this counter offer and ends up the negotiation process. Instead, he may agree the counter offer by initiating the accept messages and ends up the negotiation process by initiating the commit message.
Here, in the proposed approach, the client kicks off the template creation by filling the requirements and send the request to provider for getting approval. This is different from finding the template from provider that is closely related with the request, fill the template and send it again to the provider for acceptance. The proposed approach reduces both the average negotiation time and the number of communications between the provider and the client. In addition, it also maximizes the client’s satisfaction and SLA acceptance rate.

At the end of the SLA creation phase, the global SLA is formed for that particular tender containing the parties (i.e. resource providers) involved, SLOs, penalties etc. Once the global SLA is created, it is stored in a SLA database which is a plain XML file and the parties involved are reserved against that particular job. The SLA Creation service takes care of negotiation objectives, parties involved in the negotiation, accepted functional and non-
functional objectives, expiration time of SLAs and all these terms and conditions are stored in a XML file in a SLA repository. This repository will be referred by the SLA monitoring service, while executing the job in order to monitor the SLOs. The proposed schema for SLA is shown in the Figure 3.8. The WS-agreement acts as wrapper around this schema. The proposed schema consists of one root element called SLA tag. It consists of five child elements: JobID, ID, Expiration, SLO and Penalty. The JobID tag specifies the job id for which this SLA is created. The ID tag specifies the SLA ID. The expiration time of this SLA is specified in the Expiration tag. The requirements of each job are specified in the child element called Job, whereas the committed values of resources against this job are specified in Resource child element. Suppose if there is any violation of SLO in either side (Resource provider or user), then penalty mechanism is specified in Penalty tag. The SLA creation module consists of SLA Creation Client service and SLA Creation Server Service.

**SLA Creation Client Service** - The scheduler invokes the SLA Creation Client service with pool of Job objects that are accumulated in a Queue. The job object consists of the matched resources for that job. The SLA creation client service invokes the SLA Server Service in order to negotiate with the matched resources. The matched resources are ordered based on their deviation values. After successful negotiation with the ordered matched resource providers, the concrete SLAs are created and stored in SLA repository.

**SLA Creation Service** - The SLA Creation server service extracts the ordered matched resources and the job requirements from the Job object and starts negotiation with the matched resource providers against the functional and non-functional requirements. The functional requirement involves the
hardware and software requirements and the non functional requirement involves the response time, bandwidth, latency etc. The committed functional and non functional values are recorded as Service Level Objectives, the resource providers that are participating in the negotiation are termed as parties involved and the punishment against any violations are termed as penalty. These above mentioned terms are gathered as a concrete SLA and are stored in the SLA repository. Apart from the above mentioned terms the concrete SLA consists of jobID for which the SLA is created, an unique SLAID term and the expiration time of this SLA.

Figure 3.8 Proposed schema for SLA
**SLA Monitoring Engine** - After successful creation of agreement, the grid Meta scheduler submits the job to all the resources that are participating in that agreement. Then the monitoring component of grid middleware dynamically gets the resource/job information in periodic polling intervals and sends it to SLA Monitoring Engine. While monitoring, any over consumption in guarantee parameters of resources by the job or lesser availability of guarantee parameters than the committed value by the resource providers fires the Violation Notification. After getting the violation notification, the Event generator determines what kind of violation it is (i.e. parameters CPU or RAM) and the source of violation (user or resource provider fault) and fires the corresponding event.

In general any SLA specifies the functional as well as non-functional conditions to be met out while delivering a service to the client. To ensure this, it should monitor the provisioning of services. The monitoring process should gather the metrics to validate that the service is delivered to the client as mentioned in the SLA. The proposed SLA monitoring framework is based on publish / subscribe notification. The main key features of this proposed framework are:

- Generic extensible Level 2 (Abdul Waheed et al 2000) monitoring system built as part of GSMA.
- Support for sequential application monitoring that includes sub-processes monitoring.
- Support for multiple jobs monitoring that includes sub-processes monitoring.
- Support for parallel application monitoring that includes each slave process and its sub-processes.
• On demand dynamic sensor loading from repository based on SLOs or topics.

• Configurable local monitors thereby generate the notification messages when a process exceeds a preconfigured value (here SLO), forwards it to the notification producer from where it will be transferred to the notification consumers. This leads to reduction in message flow against its counterpart query based monitoring.

The architecture of SLA monitoring system for GSMA is shown in Figure 3.9. Since it is a level 2 monitoring system (Serafeim Zanikolas and Rizos Sakellariou 2005), it consists of following components:

• One Local Monitor (LM) per host (i.e. compute node) that gathers the resource information and application status using the sensors that are loaded based on the topics of interest (i.e. SLOs) from notification consumer. The sensors can be of host sensor, network sensor, process sensors and application sensors.

• One Main Monitor (MM) (i.e. on the head node) associates and manages multiple LMs over the compute nodes that are allocated to each job by the local resource manager. For example, if a parallel application needs three compute nodes, then the MM groups these three LMs on these nodes and treats them as single entity.

• One Monitoring Service (MS) (i.e. on the head node) that accept the queries from the notification consumers, perform the queries using MM, gathers the results of queries and send back the results to the consumer.
Here, the assumption is that the job originated as grid job and submitted to the grid resource that is available in remote site (i.e. executing site) from grid meta-scheduler. In this case, all the grid resources are installed with Globus toolkit. The normal scenario of job execution in Globus is as follows: The GRAM (Grid Resource Allocation and Management) component of executing site is integrated with the LRM (Local Resource Monitor) through LRM adapters. Once the GRAM component is configured with particular LRM, it will generate appropriate Jobmanager script. For example, if the GRAM is configured with PBS or SGE, it will generate pbs.pm or sge.pm as Jobmanager script. This job manager script is responsible for conversion of GRAM jobs into LRM specific job submission scripts. This conversion is mandatory, since the GRAM job cannot run over PBS or SGE directly. Moreover, GRAM uses the LRM to schedule and run the jobs on selected compute nodes and is mentioned in node-file.

For example, the Jobmanager script of PBS generates scheduler_pbs_job_script, scheduler_pbs_cmd_script and a scratch or temporary directory in which the abovementioned scripts are placed. Then the scheduler_pbs_job_script is submitted to PBS server. For each node in the node-file, the scheduler_pbs_job_script executes the scheduler_pbs_cmd_script to fork the process thereby initiating the application execution. These processes are identified with the help of process identifiers (PID). To monitor the SLAs, it is mandatory to track both the compute nodes that are selected for the submitted job and the PIDs generated for that specific application (i.e. job) process. The above explained normal scenario need to be modified accordingly to meet out these objectives.

Hence in the proposed SLA monitoring (refer to Figure 3.9), the tasks are divided into two parts: SLA Job monitoring (i.e. Meta-scheduler
level) and SLA monitoring (Resource provider level). The SLA monitoring is at the grid meta-scheduler level, whereas the SLA Job monitoring is at each resource provider level that includes LM, MM and sensors. These sensors, LM and MM are dynamically loaded into the executing resource based on the SLOs or topics that are present in the SLA.

For example, if an SLA put a SLO on RAM usage is less than 2 GB, only those appropriate LMs and sensors are gathered and loaded into the execution grid resource through the transporter. These sensors need PID as its input to monitor the application process against SLO or topic. To capture the PID of an application process, the name of the executable and the resource on which this process going to run are mandatory parameters. The Jobmanager scripts (Here, pbs.pm or sge.pm) are modified accordingly in way to generate a temporary shell script (i.e. getPID.sh) for capturing the PID of that application process and store it in the Globus scratch directory of that respective job. For example, if a parallel application has four processes, the utility shell script (i.e. getPID.sh) generates four PIDs and their respective execution (i.e. compute) nodes (i.e. on which node which PID is running). After this, the PIDs are forwarded to respective LMs and sensors of execution nodes, thus starts SLA job monitoring. Since, the LMs are configurable, the SLOs of that specific job are configured into LM thereby generates notification messages whenever the process consumes or provider offers more than or less than the committed value (i.e. SLO) respectively.
Figure 3.9 Proposed SLA monitoring architecture

The flow of actions in the SLA monitoring framework is as follows:

Before submitting the job to the executing resource, the executor of Meta-scheduler gets the SLA of that respective job from SLA repository. Then it extracts the SLOs or topics of interest from this SLA. Following that, it query the monitors & sensors repository to get the respective bundle of monitors and sensors. Next, it configures the LMs to these SLOs and transfers the bundle to the executing resource through the transporter of Meta-scheduler via SLA job monitor of that executing resource.
After that, the executor starts execution on the resource. Then the PIDs and computing nodes are captured as mentioned previously and are stored in a temporary PID file (i.e. pid.txt) inside the globus scratch directory of that particular job.

Once the temporary (i.e. pid.txt) PID file is generated, the LMs and sensors start monitoring the process using that PID. Suppose if a process generates sub-process, then the temporary shell utility (i.e. getPID.sh) captured that PID and rewrite the pid.txt file. Then the corresponding process’s LM and sensors are notified to monitor this new PID indicating that the previous process has been completed.

Upon monitoring, whenever there is violation against SLO, the corresponding LM generates the notification message, send it to SLA job monitoring via MM and then forward it to SLA monitoring component. The generic Level 2 monitoring usually uses the query based information gathering. To automate the SLA monitoring, it needs to be modified to notification / subscription mechanism thus eliminating lot of query messages to decide the violation. The message is sent only when there is a violation.

The SLA monitoring module implemented as three services viz: SLA monitoring client service, SLA monitoring server service and SLA job monitoring service. The SLA monitoring client service will be invoked by Executor, which in turn invokes the SLA monitoring server service. The SLA monitoring server service invokes the corresponding SLA job monitoring service of the resources on which this job is executing. The detailed implementation of SLA monitoring module is explained in the following sections.

**SLA monitoring client service** - After the successful transfer of the executables, input files, monitors and sensors bundle to the targeted resource,
the Executor starts the execution on the targeted resources. After the successful initiation of the job, the executor will invoke the SLA monitoring client service with Job ID as its argument. The SLA monitoring client service will start the SLA monitoring server service in order to capture the violation.

**SLA monitoring server service** - The SLA monitoring server service gets the job ID from SLA monitoring client service, the job object using job helper and extracts the resource(s) on which the job is executing using this job object. After identifying the resources that are actually executing the job, the SLA monitoring server service will invoke the SLA job monitoring service that is deployed into the globus container of executing resource. While invoking, the SLA monitoring server service passes the SLOs to the SLA job monitoring service.

**SLA job monitoring service** - The SLA job monitoring service place the sensors for extracting the job's runtime parameters such as RAM, Secondary storage, CPU load, bandwidth and latency. These sensors are placed inside the scratch directory automatically due to the modification done in the job manager scripts (i.e. pbs.pm and sge.pm). These runtime parameters of the job are compared against the configured committed values, which are obtained from SLA monitoring server service. Whenever the runtime values are exceeded the SLOs, only then it will be notified to the SLA monitoring server service from where it will initiate the SLA enforcement client service.

**SLA Enforcement engine** - The events generated by the Event Generator of SLA monitoring component is listened by its registered listeners of SLA Enforcement Engine. The corresponding event handler of that particular event decides what necessary action has to be taken as penalty against violation of SLA using Action Generator (refer to Figure 3.10).
In general, SLA enforcement is a technique that applies enforcement action over a violator who violates the sharing conditions (i.e. SLOs). The serious issue in enforcement is deciding who is the violator (i.e. provider or client), what kind of violation (i.e. technical or business) and what enforcement action (penalize or curative) based on the severity of violation. The violation notifications are generated when the job over consumes the resources than actually committed or the provider offers less amount of resources to the job than actually committed. In addition, violation can also be generated while the job execution may not complete within the stipulated as accepted by the provider. By analyzing the violation notification, it can be decided that who the violator is and what kind of violation it is. In the above mentioned notifications, the former is a technical violation, whereas the later is a business violation. The out of violation may ruin either the level of service that is offered to the user or reputation of the provider thus affecting the business. Then the next step is to take up the necessary actions against violations. There are two kinds of actions: penalize or curative. The penalize actions are meant for punish the violator technically or financially. The technical penalize actions try to reduce the quality of service offered to the violating client, whereas the financial penalize actions try to temporarily suspend the violating provider for ‘n’ transactions, reduction in service price, offering additional credit to affected client. The objective of corrective actions is not to penalize the violator rather to correct the system as SLA compliance.

**Figure 3.10 Proposed SLA enforcement architecture**

![Diagram of Proposed SLA enforcement architecture]
For example, a grid meta-scheduler migrates the job to some other suitable grid resource in case of system failure. The proposed approach does not implement any enforcement actions rather it just publishes the type of violation and the enforcement action. The enforcement module is implemented as SLA enforcement client service and SLA enforcement server service. The SLA Enforcement service gets invoked, when the SLA Monitoring service gets the violation notification from SLA Job Monitoring service and penalizes the user or resource provider who causes the violation (i.e. bilateral enforcement).

**SLA enforcement client service** - This service is invoked by the SLA monitoring server service and investigates the violator (user or provider) of objective and the parameter (RAM, secondary storage, load) which violates the objective. Based on the decision, it will invoke the SLA enforcement server service. The SLA enforcement server service refers the SLA from the SLA repository and decides the penalty and invokes the appropriate penalty mechanism.

**SLA enforcement server service** - The SLA enforcement server service gets the inputs from SLA enforcement client service and query the SLA repository to get the appropriate penalty mechanism to be invoked. The pluggable nature of the penalty mechanism enables that anyone can incorporate additional penalty mechanisms in the SLA enforcement engine. Based on the penalty mechanism obtained from the SLA repository, it will invoke the appropriate penalty mechanism.

In concise, the proposed PMS that is used to express the resource usage policies that include the policy editor, policy repository and policy based match-making are given in detail in the beginning. In addition, this chapter also give a detailed architecture about the SLAs that are used to record and monitor the assurances during the application execution.